STANFORD ARTIFICIAL INTELLIGENCE PROJECT August 1, 1968
MEMO AI-66

AN ALGOL-BASED ASSOCIATIVE LANGUAGE

by

J. A. Feldman and P. D. Rovner

ABSTRACT: A high-level programming language for large complex relational structures has been designed and implemented. The underlying relational data structure has been implemented using a hash-coding technique. The discussion includes a comparison with other work and examples of applications of the language. A version of this paper will appear in the communications of the ACM.

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

36

An Algol-based Associative Language

Jerome A. Feldman

Computer Science Department

Stanford University, Stanford, California

Paul D. Rovner

M.I.T. Lincoln Laboratory

Lexington, Massachusetts

An ALGOL-based associative language

1. The LEAP Language

1.a. Introduction

Associative memory, the accessing of data through a partial specification of its contents, has long been a subject of interest in Computer Science. Although it is no longer considered a panacea, associative memory has proved to be of great value in artificial intelligence, operating systems and computer aided design and is a major aspect of information retrieval. Various proposals for building large hardware associative memories have not materialized [12] and essentially all the applications to date have depended on software schemes of some sort. In this paper, we present a programming language for software associative memory systems and describe a particular scheme which seems to have several nice properties.

There are two basic problems in designing any programming system: ease of use and efficiency of execution. In this section we discuss a programming language which users have found convenient for associative processing; Section 2 describes an implementation of the language which is quite efficient over a range of problems.

The language, LEAP, is an extension of ALGOL [22] to include associations, sets and a number of auxiliary constructs. The problem of describing a programming language in a journal-size article is quite difficult, especially if the system contains new ideas. The problem is further exacerbated in this case by the fact that LEAP is really a family of languages, each adding a different set of features to the ALGOL base.

Besides the constructs described here there are forms of LEAP with matrix operations, property sets, and on-line graphics [29, 30]. Even the associative language has two compilers: one which does dynamic type checking at execution time and another (described here) which does not. This notion of fluid language specifications runs against the current emphasis on standardized languages and will be considered further.

The fluidity of the LEAP design is largely due to its implementation by means of the translator writing system, VITAL [9, 20]. Our variation of ALGOL (also described in [20]) was purposely designed with possibility of being extended, and various applications groups have tailored it to their needs. With such a system and a good linkage editor, there seems to be no advantage to standardization. We view the design of better universal language like the design of better crutches - not of direct interest to those possessing complete facilities.

This attitude toward programming languages has also influenced the style of this paper. Rather than attempt to spell out in detail every nuance of the language, we have concentrated on the major details. Although LEAP is even less context-free than other programming languages, we have expressed the form of constructs in Backus-Naur Form. The sections on semantics and pragmatics, and the examples serve to delimit the syntactically correct constructs which are meaningful. The complete formal description of LEAP in the notation of VITAL (cf. [9]) is available from the authors. This description does not include the data-structure implementation which was done separately and is described in Section 2. A translator writing system capable of automating data-structure design would be a significant contribution to the field [10].

Many of the ideas in the associative language, LEAP, have occurred before. If we consider an association to be a triple:

attribute of object is value

we see that the earliest list processing systems [18, 23] included some associative processing. By considering an associative structure as a colored, directed graph we can relate our work to string and tree manipulation languages. Both these developments have been recently surveyed in [3]. Another field in which associative structures have played a central role is information retrieval [5]. Sets have appeared in simulation languages [32] among others.

The LEAP system is a continuation [7, 26, 27] of our earlier work on associative processing. The new problems attacked were the development of a convenient language for handling complex collections of associations and the extension of the hash-coding structure techniques to a paging system. We were also interested in the implications of LEAP as a non-procedural language; the sequencing of associative retrieval statements is decided by the translator and the complexity of this process (cf. [13]) portends a difficult future for non-procedural systems. The LEAP system has been running on the Lincoln Laboratory TX-2 since early 1967 and has been used in a number of applications, some of which will be discussed in Section 3.

1.b. Data types

The basic objects of manipulation in the system are the item and an ordered triple of items called an association, a relation or just a triple. In addition to items, we introduce the auxiliary notions of set, itemvar, and local variables.

1.b.1. Syntax

```
<simple type> ::= real | integer | boolean
<algebraic type> ::= <simple type> | <simple type> array
<leap type> ::= item | itemvar | local
<type> ::= <algebraic type> | <leap type> | set |
               <algebraic type> <leap type>
```

examples:

set	sons
real item	pi
local	x
boolean	married

1.b.2. Semantics

A variable of algebraic type behaves exactly like an ALGOL variable. An item is a symbol, like a LISP [16] atom, which can be manipulated by the system. If an item is declared with an algebraic type, it will also have an associated datum of that type. Thus there are some simple variables which can be used only algebraically, some which can be used only symbolically and some which can be used in either manner. A set is an unordered collection of items containing at most one occurrence of any item.

An itemvar is a variable whose value is an item (it is a reference [36, 37] to an item). A local is an itemvar which obeys special binding rules and is used only in a restricted set of contexts (cf. Section 1.d; the earliest use of locals was in the string processing languages [3]). If an itemvar (local) is declared to have an algebraic type, the item which is the value of the itemvar (local) is assumed to have a datum of that type in algebraic operations.

1.b.3. Pragmatics

The division of variables into algebraic, symbolic and combination types is for efficiency reasons. The combination variable requires twice as much space as either of the others and produces slower code than the algebraic variable. Sets are implemented by a linked list ordered by the internal code of the items; this allows the system to carry out, e.g., set comparisons in time proportional to the sum of the size of the sets rather than their product.

There are a variety of ways to organize levels of reference within a programming system. LISP, for example, allows any level of referencing, but gives the user the responsibility for keeping things straight. ALGOL is essentially a uniform single-level system. In ALGOL 68 [33], the system accounts for any level of reference, but imposes rigid conventions on their use. The system adopted here provides three levels: itemvars and locals, items, and algebraic data. More elaborate referencing structures can be built using associations, but are the user's responsibility.

1.c. Expressions

The additional expressions required for LEAP are divided into those which yield algebraic values and those whose values are leap types.

1.c.1. Syntax

```
<additional algebraic expression>::=datum(<item expression>)|
                                    count(<set expression>)
<additional boolean expression>::=<item expression> ∈ <set expression> |
                                  <set expression> ⊂ <set expression> |
                                  <set expression> = <set expression> |
                                  istriple (<item expression>)
<item expression>::= <item> | <itemvar> | <local> | (<triple>) | any |
                     new (<algebraic expression>) | <selector> <item expression>
<set expression>::= ∅ | <set> | {<item expression list>} |
                    <set expression> <set operator> <set expression> |
                    <item expression> <associative operator> <associative
                                                              expression>
<associative expression>::= <item expression> | <set expression>
<triple>::= <item expression> · <associative expression> ≡ <associative
                                                              expression>
<set operator>::= ∩ | ∪ | -
<associative operator>::= · | ' | *
<selector>::= first | second | third
```

examples: count (sons) - datum (bill) + 3

{ bill, tom } ⊂ (sons ∪ father'don)

number · (part · hand ≡ finger) ≡ new (5)

1.c.2. Semantics

Any <item expression> yields an item as its value. If the item has an associated algebraic datum then datum will yield its value, otherwise datum is undefined. Any <set expression> yields a set as its value. The value of count is the number of items (≥ 0) in the set. The boolean relations, member (∈), contained (⊂) and set equality (=) have their usual meanings. The predicate istriple is true if its argument evaluates to an item which is a parenthesized triple (cf. next paragraph).

Either an itemvar or a local can be used wherever an item can. There is a forcing convention [36] which uses the value (an item) of the local or itemvar in these cases. The reserved item any can only appear in triples which are not operands of make (cf. 1.d.2). The unary operator new causes a new item of the type of the <algebraic expression>. A triple enclosed in parentheses is also an <item expression>; a position (first, second, or third) of such an <item expression> can be accessed by a <selector>.

example:

number · (part · hand ≡ finger) ≡ new (5)

In the construction of this triple, the system

1) forms a triple: part · hand ≡ finger
2) creates an item from the triple in 1)
3) creates a new integer item initialized to 5
4) forms the triple from number and the items of 2) and 3)

A <set expression> can be either a declared set, the empty set ∅ or a bracketed list of <item expression>s. The set operations of union, intersection and set difference are defined in the usual way. The final alternative <item expression> <associative operator> <associative expression> of <set expression> is specified in terms of the universe of triples. If the two operands are both <item expression>s with values P and Q then

P · Q is the set of items {X} such that P · Q ≡ X
is in the universe of triples.

P ' Q is the set of items {X} such that P · X ≡ Q
is in the universe of triples.

P * Q is (P · Q) ∪ (P ' Q)

If the associative expression has a set S as its value, the value of P·S is the union of P·Q for all Q in S; P'S, P*S are defined analogously.

We will discuss examples of these constructs after introducing statements in Section 1.4.

1.2.3. Pragmatics

The use of items could be avoided by adding another forcing convention to the translator; our decision to include items was meant to impose a discipline we believe to be important for effective use of LEAP. The implementation of triples is discussed in the second half of this paper; sets are discussed briefly in [illegible]. We were led to include the various set operations after noticing that associations are naturally set-valued and that all the set manipulation routines would be needed in the system in any case.

1.d. Statements

There are relatively few new statement types in LEAP because the control structure of ALGOL was largely adequate for our purposes.

1.d.1. Syntax

```
<additional statement>::= <set statement> | <associative statement>
                          <loop statement>
<set statement>::= put <item expression> in <set> |
                   remove <item expression> from <set> |
                     <set>  ← <set expression>
<associative statement>::= make <triple> | erase <triple> |
                           delete <item expression> |
                           <itemvar> ← <item expression>
<loop statement>::= foreach <local> in <set expression> do <statement> |
                    foreach <associative context> do <statement>
<associative context>::= <element> | <associative context> and <element>
<element>::= <admissable triple> | <boolean expression> |
                           <local> in <set expression>
```

examples:

```
put tom in sons
sons ← sons U {tom}
make father · tom ≡ bill
foreach x in sons do datum(x) ← datum(x)+2
foreach father · x ≡ bill do put x in sons
```

1.d.2. Semantics

The set assignment statement copies the value of the <set expression> and assigns the result to the set variable. The put (remove) operation is simply a more efficient way of doing union (subtraction) of a single element. The make (erase) statements cause a new triple to be added to (subtracted from) the universe of triples (cf. Section 2.b. for details). An item which was created by a new expression can be destroyed by delete. The internal identifier associated with this item will be reassigned and it is the user's responsibility to assume that there are no uses of a deleted item.

The <loop statements> are the raison d'etre for the entire system and will be considered in some detail. The first alternative describes the loop over the elements of a <set expression>. The <set expression> is evaluated once and the <statement> is executed once for each member of the resulting set. The local is the loop variable and is assigned the successive elements of the set; it is treated as an itemvar within the <statement>.

The loop statement over an <associative context> is much more complicated. An associative context determines a set of simultaneous relational equations which the system must solve to determine the values of the loop variables. A loop variable is a local appearing in the associative context which has not been bound in an enclosing loop. The set of values for each loop variable is determined once at the beginning of the loop and the <statement> executed repeatedly with the loop variables bound to each value in turn. The construct <admissable triple> is meant to convey the fact that certain syntactically correct <triple>s are not meaningful in an <associative context>. The compiler detects the following cases.

a) the triple contains no unbound local

b) the triple contains three unbound locals

c) the unary operator new appears in a triple.

The <boolean expression> and <set expression> elements of an associative context serve to restrict the set of values which can be assigned to a loop variable. We will discuss the processing of a loop statement with the aid of examples:

1) foreach x in sons do datum(x) ← datum(x)+2

This is a loop over a <set expression> which is simply a set sons. If x has been declared integer local, then the effect of 1) will be to increase by 2 the datum associated with each item in sons .

2) foreach father · x ≡ bill do put x in sons

We assume that father and bill are items, x is a local and sons is a set; the algebraic type of any of these variables is irrelevant. The system first computes the set of items $\{o_1, \ldots o_k\}$ which appear in a triple:

$$\text{father} \cdot o_i \equiv \text{bill} \ .$$

Now the body of the loop is executed k times, once with each o_i as the value of x (considered to be an itemvar within the body). If sons was initially empty it will be precisely the set $\{o_1, \ldots o_k\}$ at the end of the execution of 2). If k=o, the body of the loop is not executed.

In this simple case the loop statement could be replaced by

2') sons ← sons ∪ father' bill .

3) foreach father · x ≡ bill and sex · x ≡ male do put x in sons

This statement is much the same as 2) except that the set of values for x is made up of these items satisfying both associations. This keeps bill's daughters out of the set sons. Notice that the order in which the two relations are processed can be expected to have a marked effect on performance, i.e. there are probably many more males than children of bill. This situation will be discussed in Section 3.

```
4) (a)  foreach father · father · x ≡ z do make
                                grandad · x ≡ z
   (b)  foreach father · x ≡ y and father · y ≡ z do
                            make grandad · x ≡ z .
```

The statements (a) and (b) above are entirely equivalent, the compiler actually will transform (a) into (b) supplying a dummy local. Assuming x,y,z are unbound locals, the statement (b) requires solving for three loop variables. It is clearly inadequate to solve for each local independently; the system must form a n-tuple of items (a correspondence) which satisfies the associative context. In this case the n-tuple is an 3-tuple x,y,z. The correspondences are computed in advance and the body of the loop executed once for each correspondence. For example if the triples of the universe were

```
father · tom    ≡ bill
father · pete   ≡ tom
father · bill   ≡ don
father · george ≡ clyde
```

the statement (b) would yiel. correspondences

```
tom, bill, don
pete, tom, bill
```

and after the execution of (b) the universe would also contain the triples

grandad · tom ≡ don

grandad · pete ≡ bill

1.d.3. Pragmatics

The implementation of sets was discussed briefly in Section 1.b.3. The implementation of make and erase depend heavily on the structure and are discussed in Section 2.b.The loop statement over <set expression> is actually a special case of the<associative context>loop, but is implemented separately for efficiency. The associative loop is by far the most complex construct in LEAP. The formation of correspondences entails building many partial correspondences which later must be discarded. This is, in itself, a significant data structure problem and has been implemented with a hash-coded triple scheme associating a correspondence number, a local, and an item. This, combined with the order of processing considerations mentioned in example 3 above make the compilation of the associative loop statement decidely non-trivial. This problem is discussed briefly in Section 2 and at great length in Hilbing's dissertation [13].

1.e.1. Sample programs

```
begin  comment  this program will find the polygon
       in some figure which has the largest perimeter
       and will write its name, number of sides and its
       perimeter.  We assume the number of sides of a
       polygon is stored as its datum and the length
       of a line is stored as its datum, the data base is
       assumed to be built in an outer block;
   begin
       item  part,side,type,polygon;
       real  sum,max;
       local  x;
       real  local y;
       integer  itemvar  largest;
        itemvar  figure;
         read(figure);
         max ← 0;
       foreach part · figure ≡ x  and  type · x ≡ polygon  do
         begin  sum ← 0;
           foreach  side · x ≡ y  do
               sum ← sum + datum(y);
               if sum ≥ max then
                   begin largest ← x; max ← sum end
           end;
           write(largest,datum(largest),max)
   end
end
```

1.e.2.

```
begin  comment  this program will determine if mary is related
          to joe by virtue of the fact that mary's paternal aunt
          is married to joe's paternal uncle and, if so, will
          record that fact;

       begin
          item father,mary,joe,sex,married,male,related,reason;
          local x,y;
          set uncles;
          boolean switch;
               uncles ← ∅;  switch ← false;

          foreach father·x  ≡ father · father · joe and
               sex·x·≡ male do put x in uncles;
          foreach father' father · father · mary ≡ x and
                     married*(x) ≡ y and y in uncles do
                     begin
                     make related · mary ≡ joe;
                     make reason · (related · mary ≡ joe)
                       ≡ (married · x ≡ y);
                       switch ← true;
                     end;
       write (switch)
    end
end
```

2. The LEAP data-structure

The most important construct in LEAP is the <associative context> which implicitly specifies a collection of ITEMs. In this section we present a rather elaborate two-level storage scheme which was designed to facilitate the processing of associative contexts. The problem of retrieving the objects from memory which are related to a given object has been a central concern in list processing systems since their inception.

The early list-processing languages (e.g. LISP [18], IPL [23]) achieved a form of associative memory through the use of property (description) lists. This enabled one to specify the name of a property (attribute) and an object from which the system would retrieve the value or list of values. This facility freed the user from remembering which ordinal number he had mentally associated with a property. The property list was implemented as a list of (property, value) pairs linked to the object. When a retrieval was to be made, the system would search the property list of the speicified object for the specified property and would return the associated value. Although this feature of list processing systems is heavily used, there are serious problems with it from both a usage and an implementation standpoint. All of the more recent attempts to produce associative retrieval on conventional computers may be viewed as attempts to solve one or both of these problems.

The problem with property lists from a user's point of view was that they are one-way. If one has stored "SON(JOHN)=DON" there is no direct way to find the X such that "SON(X)=DON". This problem became particularly bothersome in computer graphics and led to the development of languages such as L^6 [15], CORAL [33, 34] AED [28], APL [7] and ASP[16] which have recently been surveyed [38]. These languages automatically provided two-way associations and, for reasons described in the next paragraph, replaced the property list with a block (record) of continuous storage.

The abandonment of the property list was a victory for economy over flexibility. The property list requires two cells per association and requires searching at each retrieval. The idea of using a continuous block of storage for the property list is not generally attributed to Ross [27]. The idea is simple: by assigning a small integer to each attribute (e.g. SON ~ 5) one can achieve rapid retrieval and still only use one cell per association. The usual implementation is to place the address of the top of the block (internal name of the object) in an index register and assemble a load or store instruction having that index field and having the attribute number in the address field. This scheme is efficient and has been used in a great variety of systems, but it too, has drawbacks.

The first difficulty is that the size of the continuous blocks must be specified in advance. If a new attribute is applied to an object at least a partial recompilation is required. A further difficulty occurs in scheduling the numbers assigned to each attribute; there are three approaches to this problem:

a) Eliminate symbolic attribute names; this requires the user to remember the relative location of each attribute, but does not require any extra space, [34] and to some extent [7].

b) Require that the type of block resulting from every associative retrieval be computable at translation time [36, 37].

This need not always require extra space but makes the system impractical for the type of applications considered here. For example, the APL system [7] was developed because of the inadequacy of PL/I for computer graphics.

c) Assign a unique computation to each attribute name. This allows an attribute to apply to any type of object, but impose a scheduling problem on the user which usually results in wasted space [15].

In addition, all of these schemes share the problem that the entire block of storage for an item must be allocated on the first use of that item.

The data structure schemes discussed above were designed for problems for which the associative structure rather simple and is contained in main memory. More recently, there have been attempts to extend these systems to a paging environment [2, 5, 7]. The only previous work on system capable of handling all seven associative primitives (cf. Section 3.b) and large data bases have been in query languages and information retrieval [6]. The thrust of these efforts has been rather different; they have generally emphasized data bases too large for secondary storage and have allowed for much slower response rates.

2.b. The hash-coded associative memory scheme presented here is an attempt to solve the associative processing problem for a large range of applications. Hash-coding (scrambling) is a well known technique for processing symbol tables, etc. [17, 19] and there was early speculation [21] that hash-coding could lead to efficient associative processing. The particular scheme described here is an extension of our earlier work [8, 29, 30] which is designed to work efficiently in a particular time-sharing system [11].

The first requirement that we place on an associative memory scheme is that it be capable of answering the seven retrieval requests obtained by substituting zero, one or two variables into the association.

ATTRIBUTE • OBJECT ≡ VALUE .

This requirement is an obvious extension of two-way links and has been very useful in practice. The second major requirement is that any attributes, objects and values may be combined and that an association can itself be used as an item. The third major requirement is that the time to retrieve an association should be as small as possible and should be largely independent of the total number of associations. A further requirement for the system described here is that it perform as well as possible using secondary storage within the rules of the time-sharing system.

The problem is to represent a universe of associations (triples, relations, facts) of ITEMs so as to meet the requirements of the preceding paragraph. A dictionary phase converts each ITEM to a unique integer so we can consider a universe of triples of integers (A,O,V). The seven primitive retrieval operations to be implemented are

1) (A,O,V)

2) (A,O,X)

3) (A,X,V)

4) (X,O,V)

5) (A,X,Z)

6) (X,O,Z)

7) (X,Z,V)

where X, Z denote unspecified positions (variables).

There are the additional problems that each variable may be multi-valued and that a given primitive may assign values to two variables. For example, if the universe were

(SON, DON, JOHN)

(SON, DON, JOE)

the following answers would result from primitive questions.

primitive question	result
(SON, DON, JOHN)	(SON, DON, JOHN)
(SON, JOHN, JOE)	NULL
(SON, DON, X)	X = {JOHN, JOE}
(X, DON, JOE)	X = SON
(X, DON, Z)	X = SON, Z = {JOHN, JOE}
(X, DON, DON)	NULL

The basic ideas underlying the implementation are simple. Consider first the primitive questions involving one variable (numbers 2, 3, 4). For each of these we hash-code the specified items to get an address leading to the value of the unspecified one. This address is divided into a page address and a location within the page. Multiple answers are kept on a linked list which is required to be entirely within the page. The first primitive question (A,O,V) can be answered using any of three pages; the questions involving two variables are discussed below.

The three primitive questions with two variables each involve only one specified item and so the answers could be represented by a separate list for each item. The complicated structure described below results from attempting to combine paging, hash-coding and lists in an efficient manner.

There are three types of sections each of which is associated with one of the positions A,O,V of the triple. The number of pages of each type will vary with the size of the problem. We will consider a typical A-section in some detail and then show how to extend the discussion to O and V sections.

A typical A-section contains all triples having items numbered n to $n+k$ in the A position of a triple. Let us consider the simple case of a triple $t_1 = (a,o,v)$ being placed in an otherwise empty memory. The high-order bits of a determine the proper page for t_1, the remainder of a is hashed with o to give a location in the page. At this location a cell of the following form will be constructed

o	6	CONFLICT
a-USE	v	

Now a,o,v are the elements of t_1 and "6" is the value of a tag showing the type of our cell (cf. Table 1). The CONFLICT field is used if more than one (a,o) pair hash to the same address. The a-USE field is one link in the circular list of all uses of the item 'a' in attribute position; this list provides all the answers to the primitive question (a,X,Z). The situation can get considerably more complicated as shown in Figure 1. Figure 1 depicts a segment of the storage of an A type page under the following conditions. The three pairs (a_1,o_1), (a_2,o_2) and (a_3,o_3) all hash-code to the same address, i.e. there is a three way con-

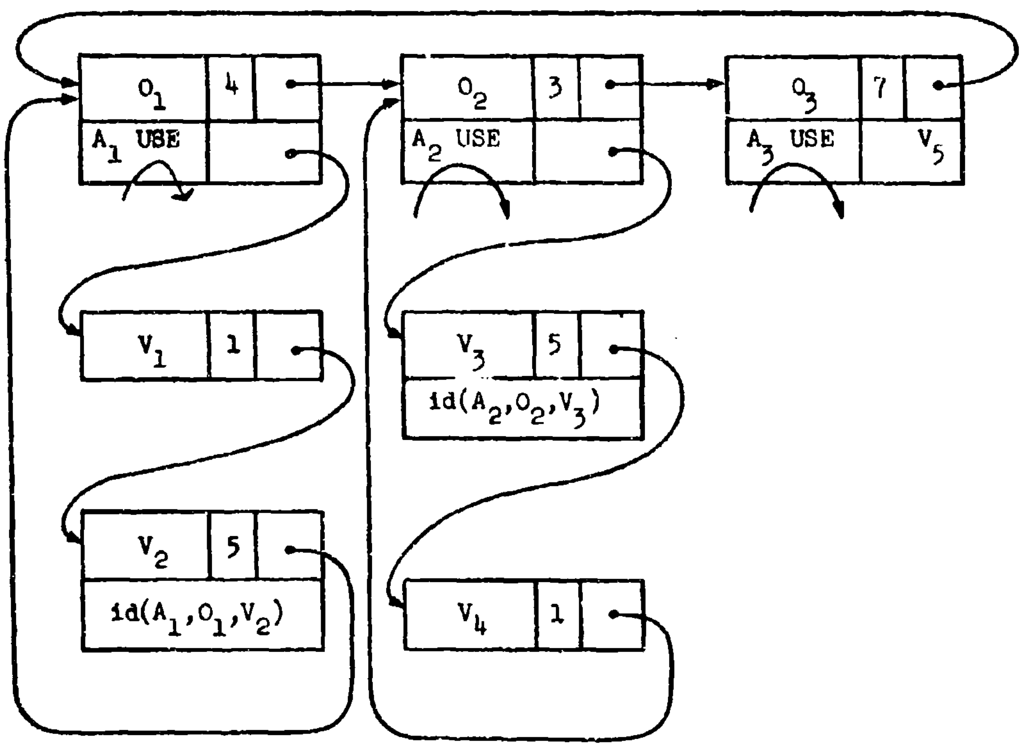

Figure 1. A Segment of the Structure on an A-type Page (cf Table 1).

flict. The associations (triples) represented are:

(a_1,o_1,v_1) (a_2,o_2,v_4)

(a_1,o_1,v_2) (a_3,o_3,v_5)

(a_2,o_2,v_3).

In addition, the triples (a_1,o_1,v_2) and (a_2,o_2,v_3) have been used as ITEMS in other associations; the internal item identifiers assigned to the respective triples are recorded in the bottom half of the type 5 cells. The meaning of each of the different cell types is given in Table 1. The situation depicted in Figure 1 is a worst case and will occur infrequently.

CELL TYPE	PURPOSE OF A CELL OF THAT TYPE
0	A two-register free block.
1	Represents a TRIPLE in a collection of TRIPLEs.
2	A one-register free block.
3	Represents a collection of TRIPLEs, in a "conflict" situation
4	Represents a collection of TRIPLEs.
5	Represents a TRIPLE which is used as an ITEM.
6	Represents a TRIPLE.
7	Represents a TRIPLE, in a "conflict" situation.

Table 1: Cell Types, and the Purpose for Each

The o and v type pages are organized in a similar fashion. Each O-page contains all uses in the "object" position of some set of items. Also on the O-page are the triples (X,o,v) for the objects of this page and for all v. Each V-page contains the uses of a set of items in the "value" position and the triples (a,X,v) for all a. Since we require that all of the multiple values of a primitive associative question be contained in one page, the pages will have to be of variable size. This is consistent with the conventions of the executive system [11], but does impose an unusual storage allocation scheme.

The storage scheme described here is designed to meet the requirements of variable size pages containing three intertwined data structures. The three structures are: The USE list, the hash-coded triples, and the free storage list. The technique used is a variation of the standard idea of allocating list space and free storage from opposite ends of a block of memory. The central idea is the "striped" page; we divide storage into classes based on the low order bits; the particular scheme now in use is described in Figure 2. Every set of eight consecutive registers is divided into two which are addressable by hash-coding, one for the heads of USE lists, and one single and two double free registers. There are many other possible arrangements and one could design a system which changed the storage layout automatically.

Let us now consider the overall operation of the storage scheme in answering the primitive associative questions. The three questions (2), (3), (4) are answered by hash coding $A \circledast O$, $V \circledast A$, and $O \circledast V$ respectively and retrieving the set of answers from the list at the resulting location on the appropriate A,V or O page. The three questions involving only one specified item, (5), (6), (7) are answered directly from the USE list for the item on the appropriate A, V or O page; the answer is a set of O-V, A-O or V-A pairs. The fully specified association (1) can be tested as either (2), (3), or (4) depending on which pages are in main memory. Thus each primitive association can be answered in one page access and without searching unless there is a hash-coding conflict. There are, in addition, commands to insert (MAKE) and delete (ERASE) triples from the universe. The system also creates dummy ITEMs for triples used in other triples.

As the reader has undoubtedly discovered, the price for this generality and efficiency is storage space. Each association (triple) is represented on an A, an O and a V type page; this requires approximately twice (not thrice) the storage of our earlier scheme [8] because some of the information is implicit in the address at which the triple is stored. The major justification for this storage redundancy is that it is only wasteful of secondary storage; the main memory requirements are actually smaller than these of any other known scheme for answering all the primitive associative questions.

There are some additional considerations which make the redundant storage scheme less costly. Representing the information in three different ways makes it possible to choose different page distribution strategies for each type. The system does not normally update the three effected pages when a MAKE or ERASE is executed; it merely records the fact that the page must be updated the next time it is brought into main memory to be accessed. An obvious addition would be an UPDATE command which caused the system to do the updating when time was available.

3. Conclusion

The associative programming language, LEAP and the hash-coded associative memory system have been in active use since early 1967. Although there have been many modifications of the system, the basic design has remained intact and appears to be sound. LEAP has become the basic applications programming language for TX-2, although many problems do not require an associative version. The associative features have bound application primarily in computer graphics.

We will briefly describe some typical applications, more detailed information can be requested through the authors. One application is an interactive program for the design of integrated circuits and the layout of their masks. Another involves an interactive system for pole-zero calculations in circuit design. There is also a program for displaying constrained [33] figures, a significant non-procedural problem. There are also two large interactive programming systems which have been written in LEAP. The first is a system for synthesizing animated cartoons from a set of continuous wave-forms specifying the motion of parts of the picture [1]. The second is the recently completed implementation [31] of the two-dimensional programming language Ambit/G [4]. In addition, a graphical debugging package for associative structures has been written.

The predominance of uses in interactive graphics systems is a result of the interests of the user community. While LEAP is unabashedly a special purpose language, we anticipate applications in many areas of question answering [6, 25] and artificial intelligence. There are continuations of some of the LEAP ideas in [17] and the as yet unpublished work of Allen

Kay at Utah, Edward Sibley at Michigan and Tim Johnson at MIT, which may extend the application areas.

Our experiences with the LEAP language and data structure has suggested several possible extensions of each. The complete data structure scheme described in Section 2 has not yet been used to nearly its capacity may be unnecessarily complicated for small graphics applications. More particularly, it would be very useful to develop a scheme which did not require a page access to ascertain that an association was not in the store.

In general, the order in which an <associative context> is processed can have a profound effect on the performance of the system. For example, consider two equivalents specification of the brothers of bill.

1) father' father(bill) and sex(x) = male

2) sex(x) = male and father' father(bill)

The second version will require time (and space for correspondences) proportional to the number of males in the universe. A study of methods for optimal rearrangement of associative statements has been completed [13] and the methods developed there could be incorporated in the compiler.

In the associative language, as in most high-level languages the full form of certain constructs (e.g. <loop statement>) becomes tedious; the system could benefit from a simple macro processor [10]. The system also suffers from the lack of item and set functions, although procedures which have the same effect are available. At a more basic level one would like the ability to manipulate entire correspondence structures (cf. Section 1.d.2). Finally, a truly complete associative system would include a facility for implicitly solving for associations which are not explicitly represented in the structure; this requires a theorem proving program as part of the access mechanism.

The study of software associative processing is expanding rapidly and there are a large number of associative languages under development. Most of these are less ambitious than LEAP, but the proposals for graphical (two-dimensional) languages [4, 32] deserve some mention. An associative structure is quite-naturally looked upon as a colored directed graph and it seems that associative processing is a natural area for graphical languages. After an early attempt to extend [34] to an associative language, we decided that this was not feasible with current technology. Ironically, the only currently available graphical language was implemented in LEAP [31]. It is too early to predict the usefulness of this system and, in any event, the problems of associative processing will be with us for some time to come.

References

1. Baecker, R. M. A study of the on-line computer aided generation of animated displays, Ph.D. Dissertation, Mass. Inst. Tech., 1968.

2. Bobrow, D. G., Murphy, D. L., "Structure of a LISP system using two-level storage," Comm. ACM, 10 (March 1967), 155.

3. Bobrow, D. (Ed.) Symbol manipulation languages and techniques, North Holland, 1968.

4. Christensen, C. "An example of the manipulation of graphs using the AMBIT/G programming language," Proc. Symp. Interactive Systems in Experimental Applied Math., Washington, D.C., 1967.

5. Cohen, J. "A use of fast and slow memories in list processing languages," Comm. ACM 10 (Feb. 1967), 82-86.

6. Cuadra, C. A. Annual Review of Information Science and Technology, New York, John Wiley, 1966.

7. Dodd, G. G. "APL a language for associative data handling in PL/I," Proc. FJCC, 1966.

8. Feldman, J. A. "Aspects of associative processing," M.I.T. Lincoln Laboratory Technical Note 1965-13, April, 1965.

9. Feldman, J. A. "A formal semantics for computer languages and its application in a compiler-compiler," Comm. ACM 9 (Jan. 1966), 3.

10. Feldman, J. A. and D. Gries "Translator writing systems," Comm. ACM 11 (Feb. 1968), 77-113.

11. Forgie, J. W. et al. "A time and memory sharing executive program," Proc. FJCC, Las Vegas, 1965.

12. Hanlan, A. E. "Content addressable and associative memory systems," IEEE Vol. EC015, No. 4 (Aug. 1966), 509-521.

13. Hilbing, F. J. The analysis of strategies for paging a large associative data structure, Ph.D. Dissertation, Industrial Engineering, Stanford University, 1968.

14. Iturriaga, R., Standish, T. A., Krutar, R. A. and Early, J. C. "Techniques and advantages of using the formal compiler writing system FSL to implement a formula ALGOL compiler," Proc. AFIPS 1966 SJCC, Vol. 28, pp. 241-252.

15. Knowlton, K. C. "A programmers description of L^6," Comm. ACM 9 (Aug. 1966), 616-625.

16. Lang, C. A. and J. Gray, "ASP-A ring implemented associative structures package," Comm. ACM ? , 1968.

17. Laurance, N. "A compiler language for data structures," preprint, Ford Motor Company, Dearborn, Michigan, 1968.

18. McCarthy, J., et al. Lisp 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Massachusetts, 1962.

19. Maurer, W. D. "An improved hash code for scatter storage," Comm. ACM 11 (Jan. 1968), 35-38.

20. Mondshein, L. F. "VITAL compiler-compiler system reference manual," M.I.T. Lincoln Laboratory Technical Note 1967-12, February, 1967.

21. Morris, R. "Scatter storage techniques," Comm. ACM 11 (Jan. 1968), 38-44.

22. Naur, P. et al. "Revised report on the algorithmic language ALGOL 60," Comm. ACM 6 (Jan. 1963), 1-17.

23. Newell, A. (Ed.) Information processing language - V manual, Prentice-Hall, Englewood Cliffs, N. J., 1961.

24. Newell, A. "A note on the use of scrambled addressing for associative memories," unpublished paper, December, 1962.

25. Raphael, B. "Sir, a computer program for semantic information retrieval," Proc. FJCC, 1964.

26. Roberts, L. G. "Machine perception of three-dimensional solids," M.I.T. Lincoln Laboratory Technical Report No. 315, May, 1963.

27. Ross, D. T. "A generalized technique for symbolc manipulation and numerical computation," Comm. ACM 4 (March 1961), 147-150.

28. Ross D. T. "The AED approach to generalized computer aided design," Proc. ACM 22nd Natl. Conf. 1967, pp. 367-385.

29. Rovner, P. D. and J. A. Feldman, "An associative processing system for conventional digital computers," M.I.T. Lincoln Laboratory Technical Note 1967-19, April, 1967.

30. Rovner, P. D. and J. A. Feldman, "The LEAP language and data structure," M.I.T. Lincoln Lab., January 1968. In abbreviated form, Proc. IFIP Conf. 1968.

31. Rovner, P. D. "An Ambit/G programming language implementation," Lincoln Laboratory, M.I.T., June 1968.

32. Savitt, D., H. H. Love, R. E. Troop, "ASP, a new concept in language and machine organization," Proc. Spring Joint Comp. Conf. 1967.

33. Sutherland, I. E. "Sketchpad: a man-machine communication system," Proc. SJCC, 1963.

34. Sutherland, W. R. "On-line graphical specification of procedures," M.I.T. Lincoln Laboratory Technical Report No. 405, May, 1966.

35. Teichrow, D. and Lubin, J. F. "Computer simulation - a discussion of the technique and comparison of languages," Comm. ACM 9 (Oct. 1966), 727-741.

36. Van Wijngaarden, A. (Ed.) Draft report on ALGOL 68, MR 93, MATHEMATISCH CENTRUM, Amsterdam, 1968.

37. Wirth, N. and C.A.R. Hoare, "A contribution to the development of ALGOL," Comm. ACM 9 (June 1966).

38. Gray, J. C. "Compound data structures for computer aided design," Proc. ACM Natl. Conf., 1967.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Artificial Intelligence Project Stanford University Computer Science Department	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

An Algol-Based Associative Language

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
A.I. Memo

5. AUTHOR(S) *(First name, middle initial, last name)*

J. A. Feldman, P. D. Rovner

6. REPORT DATE August 1, 1968	7a. TOTAL NO. OF PAGES 34	7b. NO. OF REFS 38
8a. CONTRACT OR GRANT NO. SD-183 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) AI-66 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. DISTRIBUTION STATEMENT

Statement No. 1 - Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

A high-level programming language for large complex relational structures has been designed and implemented. The underlying relational data structure has been implemented using a has-coding technique. The discussion includes a comparison with other work and examples of applications of the language. A version of this paper will appear in the communications of the ACM.

DD FORM 1 NOV 65 1473

Security Classification

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
associative						
relational structures						
programming languages						
non-procedural						

Security Classification